COMPUTING SCIENCE

SONCraft: A Tool for Construction, Simulation and Verification of Structured Occurrence Nets

Authors: Bowen Li, Maciej Koutny and Brian Randell

TECHNICAL REPORT SERIES

No. CS-TR-1493 January 2016

TECHNICAL REPORT SERIES

No. CS-TR-1493 January, 2016

SONCraft: A Tool for Construction, Simulation and Veri_cation of Structured Occurrence Nets

Abstract

Structured occurrence nets (SONs) are a Petri net based formalism for portraying the behaviour of complex evolving systems. The concept extends that of occurrence nets - a formalism that can be used to record causality and concurrency information concerning a single execution of a system. In SONs, multiple occurrence nets are combined by various types of relationships. In particular, relationships are included that enable the representation of dependencies between communicating and evolving sub-systems. In this paper, we introduce a tool for editing, simulating, and analysing SONs. The present version deals with three of the various types of abstractions that have been defined for SONs.

Bowen Li, Maciej Koutny and Brian Randell
School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, United Kingdom
fbowen.li, maciej.koutny, and brian.randellg@ncl.ac.uk

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

SONCraft: A Tool for Construction, Simulation
and Veri_cation of Structured Occurrence Nets
Bowen Li, Maciej Koutny and Brian Randell
School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, United Kingdom
fbowen.li, maciej.koutny, and brian.randellg@ncl.ac.uk

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1493

Abstract

Structured occurrence nets (sons) are a Petri net based formalism for portraying the behaviour of complex evolving systems. The concept extends that of occurrence nets - a formalism that can be used to record causality and concurrency information concerning a single execution of a system. In sons, multiple occurrence nets are combined by
various types of relationships. In particular, relationships are included that enable the representation of dependencies between communicating and evolving sub-systems. In this paper, we introduce a tool for editing, simulating, and analysing sons. The present version deals with three of the various types of abstractions that have been defined for sons.

About the authors

Bowen Li is currently a Senior Research Associate in the Advanced Model-Based Engineering and Reasoning (AMBER), School of Computing Science at Newcastle University. He is working on the EPSRC funded project UNCOVER (UNderstanding COmplex system eVolution through structurEd behaviours). An overall goal of UNCOVER is to develop a rigorous methodology supported by a toolkit based on structured occurrence nets, in order to provide an effective approach to acquiring and exploiting behavioural knowledge of a complex evolving system.
Maciej Koutny is currently a Professor of Computing Science in the School of Computing Science, Newcastle University. He received his MSc (1982) and PhD (1984) in Applied Mathematics from the Warsaw University of Technology, Poland. In 1985 he joined the then Computing Laboratory of the University of Newcastle upon Tyne to work as a Research Associate. In 1986 he became a Lecturer in Computing Science at Newcastle, and from 1994 to 2000 he held an established Readership at Newcastle University. His research interests centre on the theory of distributed and concurrent systems, including both theoretical aspects of their semantics and application of formal techniques to the modelling, synthesis and verification of such systems; in particular, model checking based on net unfoldings. He has also investigated non-interleaving semantics of priority systems, and the relationship between temporal logic and process algebras. He has been working on the development of a formal model combining Petri nets and process algebras as well as on Petri net based behavioural models of membrane systems.
Professor Brian Randell graduated in Mathematics from Imperial College, London in 1957 and joined the English Electric Company where he led a team that implemented a number of compilers, including the Whetstone KDF9 Algol compiler. From 1964 to 1969 he was with IBM in the United States, mainly at the IBM T.J. Watson Research Center, working on operating systems, the design of ultra-high speed computers and computing system design methodology. He then became Professor of Computing Science at the University of Newcastle upon Tyne, where in 1971 he set up the project that initiated research into the possibility of software fault tolerance, and introduced the "recovery block" concept. Subsequent major developments included the Newcastle Connection, and the prototype

Distributed Secure System. He has been Principal Investigator on a succession of research projects in reliability and security funded by the Science Research Council (now Engineering and Physical Sciences Research Council), the Ministry of Defence, and the European Strategic Programme of Research in Information Technology (ESPRIT), and now the European Information Society Technologies (IST) Programme. Most recently he has had the role of Project Director of CaberNet (the IST Network of Excellence on Distributed Computing Systems Architectures), and of two IST Research Projects, MAFTIA (Malicious- and Accidental-Fault Tolerance for Internet Applications) and DSoS (Dependable Systems of Systems). He has published nearly two hundred technical papers and reports, and is co-author or editor of seven books. He is now Emeritus Professor of Computing Science, and Senior Research Investigator, at the University of Newcastle upon Tyne.

Suggested keywords

SONCraft: A Tool for Construction, Simulation and Verification of Structured Occurrence Nets

Bowen Li, Maciej Koutny and Brian Randell

School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, United Kingdom
{bowen.li, maciej.koutny, and brian.randell}@ncl.ac.uk

Abstract. Structured occurrence nets (SONs) are a Petri net based formalism for portraying the behaviour of complex evolving systems. The concept extends that of occurrence nets – a formalism that can be used to record causality and concurrency information concerning a single execution of a system. In SONs, multiple occurrence nets are combined by various types of relationships. In particular, relationships are included that enable the representation of dependencies between communicating and evolving sub-systems. In this paper, we introduce a tool for editing, simulating, and analysing SONs. The present version deals with three of the various types of abstractions that have been defined for SONs.

1 Introduction

The concept of structured occurrence nets (SONs) [7, 14, 13] is an extension of occurrence nets [3]. Occurrence Nets are directed acyclic graphs that represent causality and concurrency information concerning a single execution of a system. The SON formalism has been introduced to enable the portrayal of the behaviours of complex evolving systems. Such systems generally consist of a large number of sub-systems which may proceed concurrently and interact with each other while their behaviour is subject to modification by other sub-systems. The design and behaviour of such systems can be highly complex due to their intricate dependencies, and a large number of recordable events and system state information.

The underlying idea of a SON is to combine multiple related occurrence nets by using various formal relationships, in particular, in order to express dependencies between interacting and evolving systems. By means of these relations, a SON is able to portray a more explicit view of system evolution, involving various types of communication, system upgrades, reconfigurations and replacements, that allows one to exploit the behavioural knowledge of a complex evolving system.

Communication structured occurrence nets (CSONs) are the fundamental variant of structured occurrence nets that has the capability of providing a meaning of the synchronous interaction between communicating systems. Intuitively, a CSON combines two or more occurrence nets into a single structure

by letting them communicate via two special relationships, viz., synchronous and asynchronous communications. The former implies that a sender waits for an acknowledgement of a message before proceeding, while in the latter the sender proceeds without waiting.

Behavioural structured occurrence nets (BSONs) convey information about the evolution of individual systems. A system in BSON has a two-level view of its execution history: the structure at a lower level provides the details of its abstract behaviour represented at an upper level. The abstract (behavioural) relations between two different levels show their consistent dependencies. Figure 1 shows a simple example of BSON in a (off-line) system update. The upper level represents a version change caused by an update event. The lower level provides a detailed behaviour of the system before and after the update. The dashed lines between the two levels are used to capture the relevant relationships between the two types of behaviours. (The update portrayed is termed "offline", in contrast to an online update, such as would be exemplified in the Figure if the final state of ON_1 were also the initial state of ON_2.)

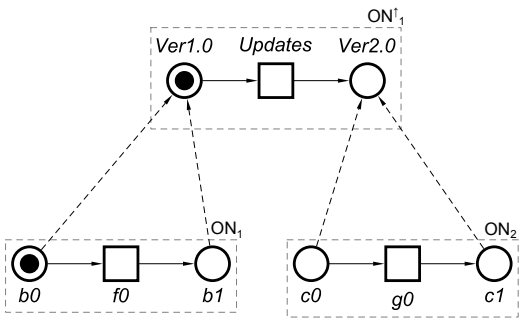

Fig. 1. A BSON example portraying (off-line) system update.

Recognising the need for a tool to support the construction and analysis of structured occurrence nets, we have developed SONCraft — an open source tool for SON visualisation, verification, and model analysis. The tool is implemented as a Java plug-in to the Workcraft platform [12] - a flexible framework for the development and analysis of Interpreted Graph Models [11]. SONCraft provides a user-friendly graphical interface that facilitates model entry, supports interactive visual simulation, and allows the use of a set of model checking tools.

The present paper has two parts. The first part discusses the basic concepts and several important properties of SONs. We propose a new causal relation for BSONs which captures the dependencies between events in different levels. Moreover, we define execution semantics of SONs which can be used for a step by step simulation. The second part of the paper outlines SONCraft and describes a set of algorithms used in the implemented analysis tools.

The rest of the paper is organised as follows. In Section 2 we recall the main notions concerning occurrence nets. Sections 3 and 4 present the concepts and properties of CSONs and BSONs, respectively. Section 5 outlines the SONCraft framework and describes additional tools that have been added for model checking and simulation. Section 6 concludes the paper.

2 Occurrence Nets

In this section, we first introduce the concept of occurrence nets, and then recall from [7] several notions and properties based on the structure of occurrence nets.

Occurrence nets are directed acyclic graphs used to record dependencies between events in a single execution of a concurrent system. One can derive an occurrence net in two different ways: (i) as a process underpinning a run of a standard Petri net, e.g., place/transition net (PT-net); or (ii) as a direct representation of an actual or imagined system's execution history (such a system may involve not only computer components, but also components and systems involving people and physical processes). Thus, only information about concurrency and causality between events and visited local states is represented, and the underlying mathematical structure is that of a partial order.

An *occurrence net* is a finite triple $\mathrm{ON} = (C, E, F)$, where C and E are disjoint sets of respectively *conditions* and *events* (collectively referred to as the *nodes*), and $F \subseteq (C \times E) \cup (E \times C)$ is the *flow* relation. The *inputs* and *outputs* of a node x are respectively defined as ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. It is also assumed that the following are satisfied:

- For all $c \in C$ and $e \in E$: $|{}^\bullet c| \leq 1$, $|c^\bullet| \leq 1$, $|{}^\bullet e| \geq 1$, and $|e^\bullet| \geq 1$.
- The causality relation $\prec$ over E is acyclic, where $e \prec f$ if there is $c \in C$ with $c \in e^\bullet \cap {}^\bullet f$.

$M_0^{\mathrm{ON}} = \{c \in C \mid {}^\bullet c = \emptyset\}$ is the *initial marking* of ON (in general, a *marking* is any set of conditions).

To summarise, an occurrence net is a direct acyclic graph, which consists of conditions, events, and arcs. Each arc runs from a source condition to a destination event, or from a source event to a destination condition; the source node (condition or event) is termed an input of the destination node (event or condition respectively), and the destination node is termed an output of the source node. Each condition has at most one input event and at most one output event; and each event has at least one input condition and at least one output condition. Moreover, the set of all conditions with no input events is the initial marking (denoted by $Init$ or M_0^{ON}), and the set of all conditions with no output events is the final marking (denoted by Fin).

Two nodes, x and y, are causally related if $(x, y) \in F^+$ or $(y, x) \in F^+$; otherwise they are concurrent. A *co-set* is a set $B \subseteq C$ comprising pairwise concurrent conditions. Moreover, a *cut* is any maximal (w.r.t. $\subseteq$) co-set.

Next we recall notions and properties concerning occurrence nets which are useful in the rest of the this paper. In this paper, if a variant of SONs is clear from the context, we will write the corresponding initial marking as M_0.

Given an initial marking, the execution of an occurrence net proceeds by the occurrence (or firing) of sets of events. The firing rule below specifies the conditions under which a marking enables a set of events (called a *step*), and how the firing of the events changes the current marking.

Definition 1 (ON firing rule). *Let* $\text{ON} = (C, E, F)$ *be an occurrence net,* M *be a marking, and* U *be a step of* ON.

1. U *is* ON*-enabled at* M *if* ${}^\bullet e \subseteq M$*, for every* $e \in U$.
2. *If* U *is* ON*-enabled at* M*, then* U *can be fired and produce a new marking* M' *given by* $M' = (M \setminus {}^\bullet U) \cup U^\bullet$*, where* ${}^\bullet U = \bigcup_{e \in U} {}^\bullet e$*, and* $U^\bullet = \bigcup_{e \in U} e^\bullet$.

   This is denoted by $M[U\rangle_{\text{ON}} M'$.

A *step sequence* of ON is a sequence $\lambda = U_1 \ldots U_n$ $(n \geq 0)$ of steps such that there exist markings $M_1, \ldots, M_n$ satisfying:

$$M_0^{\text{ON}}[U_1\rangle_{\text{ON}} M_1, \ldots, M_{n-1}[U_n\rangle_{\text{ON}} M_n \ . \tag{1}$$

The *reachable markings* of ON are defined as the smallest (w.r.t. $\subseteq$) set *reach*(ON) containing M_0^{ON} and such that if there is a marking $M \in$ *reach*(ON) and $M[U\rangle_{\text{ON}} M'$, for a step U and a marking M', then $M' \in$ *reach*(ON).

Proposition 1. *(see [7]) Given a step sequence of* ON *defined by (1), we have that:*

1. *If* $i \neq j$ *then* $U_i \cap U_j = \varnothing$*, i.e., no event occurs more than once.*
2. *There is a step sequence involving all the events in* E.
3. $Fin = M_n$ *iff* $E = U_1 \cup \cdots \cup U_n$*, i.e., each event of* E *has occurred.*
4. *If* $i \geq j$ *then* $(U_i \times U_j) \cap \prec^+ = \varnothing$*, i.e., the causal predecessors of an event can never be executed after or together with that event.*

Figure 2 shows an occurrence net – conditions are represented by circles and events are represented by boxes. The initial marking is $\{c_0\}$ which is indicated by showing a token inside the starting condition. A possible step sequence is $\lambda = \{e_0\}\{e_1, e_2\}\{e_3\}$. One can observe that the corresponding sequence of markings starts with the $Init = \{c_0\}$ and ends with $Fin = \{c_5\}$. Moreover, there are five cuts: $\{c_0\}$, $\{c_1, c_2\}$, $\{c_1, c_4\}$, $\{c_3, c_4\}$, $\{c_2, c_3\}$, and $\{c_5\}$ (the dashed lines in the figure indicates three of them).

A *phase* of ON is a non-empty set of conditions $\pi \subseteq C$ such that the set $Min_\pi \subseteq \pi$ of the minimal conditions of π (w.r.t. F^+) is a cut; the set $Max_\pi \subseteq \pi$ of the maximal conditions of π (w.r.t. F^+) is a cut; and π comprises all conditions $c \in C$ for which there are $b \in Min_\pi$ and $d \in Max_\pi$ satisfying $(b, c) \in F^*$ and $(c, d) \in F^*$. Moreover, a *phase decomposition* of ON is a sequence $\pi_1 \ldots \pi_m$ of

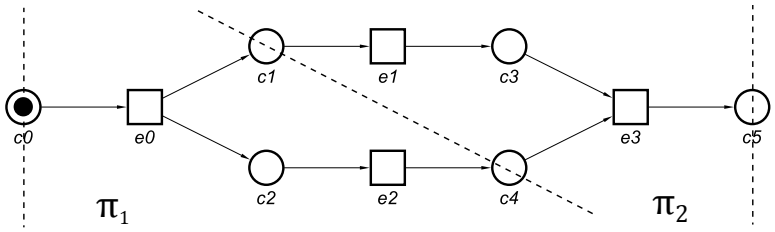

Fig. 2. An occurrence net and one of its possible phase decompositions.

phases of the occurrence net such that $Init = Min_{\pi_1}$, $Max_{\pi_i} = Min_{\pi_{i+1}}$ (for $i \leq m-1$), and $Max_{\pi_m} = Fin$.

The phase is a fragment of the ON beginning with a cut and ending with a cut which follows it in the causal sense, including all the conditions occurring between these two cuts. A *phase decomposition* is a sequence of phases from the initial state to the final state, and whenever one phase ends, its maximal cut is the start point of the successive one (minimal cut). As an example, the ON in Figure 2 has been divided into two phases by the three depicted cuts. The corresponding phase decomposition is $\pi_1\pi_2 = \{c_0, c_1, c_2, c_4\}\{c_1, c_3, c_4, c_5\}$.

3 Communication Structured Occurrence Nets

Communication structured occurrence nets (CSONs) are able to portray different kinds of communication between separate systems. It will usually be the case that if an occurrence net in fact represents the combined activity of several interacting systems, it will be beneficial to split the model into a set of component occurrence nets, and create specific devices to represent communication between the component occurrence nets (subsystems). In the model we are interested in, communication can be synchronous or asynchronous.

A CSON is composed of a set of component ONs representing separate subsystems. When it is determined that there is a potential for an interaction between subsystems, asynchronous or synchronous communication link can be made between events in different ONs via a special element called a *channel place*, portrayed graphically by a bold circle. The communication relations were represented by a directed dashed line between two events in the original definition of CSONs [7]. The notion of a channel place, which was introduced in [6], is a more flexible means of representing such relations. The new notion can be used to implement the causality expressed through the communication arcs in SONs.

Two events involved in a synchronous communication link must be executed simultaneously. On the other hand, events involved in an asynchronous communication can be either executed simultaneously, or one after the other.

Definition 2 (CSON). *A* communication structured occurrence net *(*CSON*) is a tuple*

$$\text{CSON} = (\text{ON}_1, \ldots, \text{ON}_k, Q, W)$$

such that $\text{ON}_i = (C_i, E_i, F_i)$ *for* $i = 1, \ldots, k$ *are occurrence nets (below we denote by* $\mathbf{C} = \bigcup_{i=1}^{k} C_i$, $\mathbf{E} = \bigcup_{i=1}^{k} E_i$ *and* $\mathbf{F} = \bigcup_{i=1}^{k} F_i$ *their conditions, events and arcs); Q is a set of channel places; and* $W \subseteq (\boldsymbol{E} \times Q) \cup (Q \times \boldsymbol{E})$ *are the arcs between the channel places and events. It is further assumed that:*

1. *The* ON_i*'s and Q are mutually disjoint.*
2. *The sets of input and output events of* $q \in Q$*,*

$$^\bullet q = \{e \in \boldsymbol{E} \mid (e, q) \in W\} \quad \textit{and} \quad q^\bullet = \{e \in \boldsymbol{E} \mid (q, e) \in W\}\,,$$

   belong to distinct component ON_i*'s; and moreover,* $|^\bullet q| = 1$ *and* $|q^\bullet| \leq 1$*.*
3. *The relation*

$$(\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^* \tag{2}$$

   over $\boldsymbol{E}$ is irreflexive, where:
   - $e \prec f$ *if there is* $c \in \boldsymbol{C}$ *with* $c \in e^\bullet \cap {}^\bullet f$*;*
   - $e \sqsubset f$ *if there is* $q \in Q$ *with* $q \in e^\bullet \cap {}^\bullet f$*.*

In Definition 3(2), we use the relation $\sqsubset$ (*weak causality*) to represent a/synchronous communication between two events (see [5]). Intuitively, the original causality relation $\prec$ represents the 'earlier than' relationship on the events, and $\sqsubset$ represents the 'not later than' relationship. The input and output sets of a node in CSON are also extended to include channel places with the relation W. In order to ensure that the resulting causal dependencies remain consistent, in (2) we require the acylicity of not only each component occurrence net but also any path involving $\prec$.

The initial marking M_0^{CSON} of a CSON is the union of $M_0^{\text{ON}_1}, \ldots, M_0^{\text{ON}_k}$ (in this paper we assume there is no channel place in M_0^{CSON}). In general, a marking in CSON is a set of conditions and channel places. A step in CSON is a set of events which may come from one or more component occurrence nets.

Definition 3 (CSON firing rule). *Let* CSON *be a communication structured occurrence net as in Definition 2, M be a marking, and U be a step of* CSON*.*

1. *U is* CSON-enabled *at M if* $({}^\bullet U \setminus U^\bullet) \subseteq M$*.*
2. *If U is* CSON*-enabled at M, then U can be* fired *and produce a new marking M' is given by:* $M' = (M \cup U^\bullet) \setminus {}^\bullet U$*. This is denoted by* $M[U\rangle_{\text{CSON}} M'$*.*

The *step sequences* and *reachable markings* of CSON are then defined similarly as for an occurrence net.

The firing rule above means that a step U involving synchronous behaviour can use not only the tokens that are already available in channel places at marking M, but also can use the tokens deposited there by events from U during the execution of U. In this way, events from U can 'help' each other individually

and synchronously pass resources (tokens) among themselves. Thus, in contrast to the step sequence of an occurrence net, where a step consists of a number of enabled events, the execution of a step in a CSON (i.e., $M[U\rangle M'$) may involve synchronous communications, where events execute simultaneously and behave as a transaction. Such a mode of execution is strictly more expressive than that used in ONs.

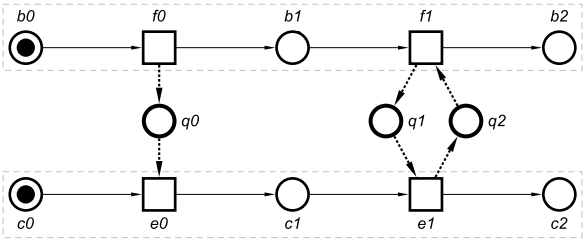

Fig. 3. A CSON with two interacting occurrence nets.

Figure 3 shows a CSON which consists of two interacting occurrence nets connected by three channel places (represented by circles with thick edges). The thick dashed lines indicate the flow relation W. The connection between events f_0 and e_0 is an asynchronous communication, which means that e_0 cannot happen before f_0. Events f_1 and e_1 are connected by a pair of empty channel places, q_1 and q_2, forming a cycle. Such a cycle does not violate CSON's acyclicity because it involves only weak causality, but the two connected events can only be executed synchronously. The channel places q_1 and q_2 will be filled and emptied synchronously when both f_1 and e_1 participate in a step being fired. Therefore, a possible step sequence of this CSON is $\lambda = \{f_0\}\{e_0\}\{f_1, e_1\}$

Definition 4 (sync-cycle). *Let* CSON *be a communication structured occurrence net as in Definition 2.*

A sync-cycle *of* CSON *is a maximal nonempty set of events* $\mathbb{S} \subseteq \mathbf{E}$ *such that for all distinct* $e, f \in \mathbb{S}$, $(e, f) \in W^+$. *The set of all sync-cycles of* CSON *will be denoted by* SC^{CSON}.

A channel place q *is* synchronous *if there exist a sync-cycle* $\mathbb{S} \in SC^{\text{CSON}}$ *such that* $q \in \mathbb{S}^\bullet \cap {}^\bullet\mathbb{S}$. *Otherwise,* q *is* asynchronous.

The notion of a sync-cycle captures the idea of a synchronous communication involving a maximal number of sub-systems. Its events graphically form a weak causal cycle connected by synchronous channel places.

We first show that there is no reachable marking which includes synchronous channel places.

Proposition 2. *Let* CSON *be a communication structured occurrence net as in Definition 2, and* Q^s *be its synchronous channel places. Then* $Q^s \cap M = \varnothing$, *for every reachable marking* $M \in reach(\text{CSON})$.

Proof. By the definition of CSON, we have $Q^s \cap M_0^{\text{CSON}} = \varnothing$. Hence, it suffices to show that if $M \in \mathit{reach}(\text{CSON})$ is such that $Q^s \cap M = \varnothing$, and M' and U are such that $M[U\rangle_{\text{CSON}} M'$, then $Q^s \cap M' = \varnothing$.

Suppose $q \in Q^s$ is such that $q \in M'$. Then there is a sync-cycle $\mathbb{S} \in SC^{\text{CSON}}$ and $e, f \in \mathbb{S}$ such that $q \in e^\bullet \cap {}^\bullet f$. By Definition 4, there is a sequence $e_0 q_0 e_1 \ldots e_{m-1} q_{m-1} e_m$ such that $e_0 = f$, $e_m = e$ and $q_i \in e_i^\bullet \cap {}^\bullet e_{i+1}$, for $i < m$. We also recall that $|q_i^\bullet| = |{}^\bullet q_i|$, for every $i < m$.

Since $q \in M'$, we have $f = e_0 \notin U$. Hence, since $q_0 \notin M$, we have $e_1 \notin U$. By proceeding k times in this way, we obtain $e_m = e \notin U$. This and $q \notin M$ means that $q \notin M'$, a contradiction. As a result, $Q^s \cap M' = \varnothing$. □

The next result implies that all events in a sync-cycle are always enabled and fired simultaneously.

Proposition 3. *Let* CSON *be a communication structured occurrence net as in Definition 2,* $\mathbb{S} \in SC^{\text{CSON}}$ *be a sync-cycle, and* U *be a step enabled at a reachable marking* $M \in \mathit{reach}(\text{CSON})$*. Then* $e \in U \Leftrightarrow f \in U$*, for all* $e, f \in \mathbb{S}$*.*

Proof. Follows from Proposition 2, Definition 4, and the definition of an enabled step. □

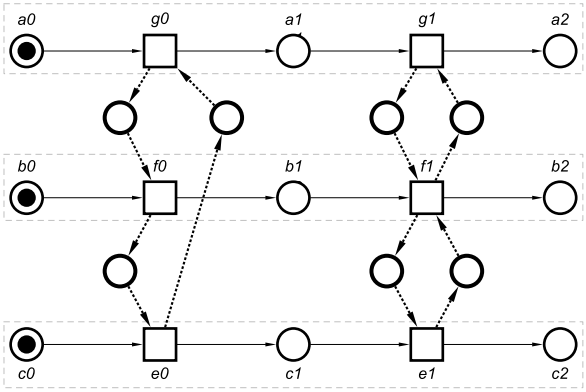

Fig. 4. Three occurrence nets that are synchronous with each other.

Consider the CSON in Figure 4. One can observe there are two sync-cycles Sync-cycle $\{e_0, f_0, g_0\}$ in fact is composed of three asynchronous communications. The communication in any of two events is asynchronous. However, all three events can only fire in a single step. The run for the sync-cycle $\{e_1, f_1, g_1\}$ is also simultaneous. Although it consists of two 'component' synchronous interactions, it is impossible to fire either of them individually [1].

[1] To simplify the representation, in rest of the paper we will use bold dashed lines without arcs to indicate any synchronous cycles linked directly between two events.

The following proposition addresses the minimal firing concerning asynchronous communication .

Proposition 4. *Let* CSON *be a communication structured occurrence net as in Definition 2, q be an asynchronous channel place, e and f be the input and output events of q respectively, M be a reachable marking, and U be a* CSON*-enabled step at M. Then $f \in U$ and $q \notin M$ implies $e \in U$.*

Proof. Suppose that $e \notin U$. From Definition 3(1), $f \in U$ implies $q \in M$, a contradiction. □

In Figure 3, if e_0 is in U and q_0 is not marked, then the occurrences of e_0 and f_0 must happen together.

Proposition 5 ([7]). *Let* CSON *be a communication structured occurrence net as in Definition 2, and $\lambda = U_1 \dots U_n$ $(n \geq 0)$ be a step sequence of* CSON*.*

1. *If $i \neq j$ then $U_i \cap U_j = \varnothing$, i.e., no event occurs more than once.*
2. *There is a step sequence involving all the events in $\mathbf{E}$.*
3. *If $i \geq j$ then $(U_i \times U_j) \cap ((\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^*) = \varnothing$, i.e., the causal predecessors of an event can never be executed after or together with that event.*

4 Behavioural Structured Occurrence Nets

Behavioural structured occurrence nets (BSONs) allow the activity of an evolving system to be modelled. They use a two-level view to represent an execution history, with the lower level providing details of its behaviours during the different evolution stages represented in the upper level view. Thus a BSON gives information about the evolution of an individual system, and the phases of the overall activity are used to represent each successive stage of the evolution of this system.

4.1 Behavioural Structured Occurrence Nets

We first recall two relations in CSON which extend the definitions of $pre(x)$ and $post(x)$ [2]. Given a CSON as in Definition 2 and $e \in \mathbf{E}$ be an event, the sets $Pre(e)$ and $Post(e)$ respectively comprise all conditions $c \in \mathbf{C}$ satisfying $(c, e) \in \mathbf{F} \circ \sqsubset^*$ and $(e, c) \in \sqsubset^* \circ \mathbf{F}$. Intuitively, the new relations capture weak causal chains passing through the events in different occurrence nets. For example, the new relationships in Figure 3 are:

$$\begin{array}{lll} Pre(f_0) = \{b_0\} & Pre(f_1) = \{b_1, c_1\} & Pre(e_0) = \{b_0, c_0\} \\ Pre(e_1) = \{b_1, c_1\} & Post(f_0) = \{c_1, b_1\} & Post(f_1) = \{b_2, c_2\} \\ Post(e_0) = \{c_1\} & Post(e_1) = \{b_2, c_2\}\ . & \end{array}$$

[2] In this section, we will use notations $pre(x)$ and $post(x)$ instead of 'dot' to represent input and output for the purpose of clarity.

We now introduce the BSON concept by using the notions above, and by generalising the definition of [7]. Below we assume that an occurrence net ON is *line-like* if $|M_0^{\text{ON}}| = 1$ and $|{}^\bullet e| = |e^\bullet| = 1$, for every event e. Such an occurrence net can be represented in a unique way by a chain $\xi_{\text{ON}} = c_1 e_1 \dots e_{l-1} c_l$ of alternating (all) conditions and (all) events satisfying ${}^\bullet e_i = \{c_i\}$ and $e_i^\bullet = \{c_{i+1}\}$, for every $i < l$.

A BSON consists of two CSONs linked by behavioural relation β. The CSON which all of whose component occurrence nets are line-like and all the conditions are the end points of β, is the upper level net that represents the evolution of a system (denoted by $\text{CSON}^\uparrow$). While the other CSON is the lower level net that represents the detailed behaviour of the system. The behavioural relation β connecting the two levels is used to provide dependencies between the evolution and detailed information of the system. In such a structured view the upper part provides the necessary information for the desired sequencing of the occurrence nets (which are called phases) in the lower part.

Definition 5 (BSON). *Let* CSON *be a communication structured occurrence net as in Definition 2, and* $\text{CSON}^\uparrow = (\text{ON}_1^\uparrow, \dots, \text{ON}_m^\uparrow, Q^\uparrow, W^\uparrow)$ *be another (disjoint) communication structured occurrence net such that* $\text{ON}_i^\uparrow = (C_i^\uparrow, E_i^\uparrow, F_i^\uparrow)$ *is line-like, for* $i \leq m$. *Moreover, let* $\mathbf{C}^\uparrow = \bigcup_{i=1}^m C_i^\uparrow$, $\mathbf{E}^\uparrow = \bigcup_{i=1}^m E_i^\uparrow$, *and* $\mathbf{F}^\uparrow = \bigcup_{i=1}^m F_i^\uparrow$.

A behavioural structured occurrence net *(or* BSON*) is a tuple*

$$\text{BSON} = (\text{CSON}, \text{CSON}^\uparrow, \beta)$$

such that $\beta \subseteq \mathbf{C} \times \mathbf{C}^\uparrow$.
It is assumed that the following hold:

1. *For every* ON_i, *there exists exactly one* $\text{ON}_j^\uparrow$ *satisfying* $\beta(C_i) \cap C_j^\uparrow \neq \varnothing$.
2. *For every* $\text{ON}_j^\uparrow$ *represented by a chain* $\xi_{\text{ON}_j^\uparrow} = c_1 e_1 \dots e_{l-1} c_l$, *the sequence* $\pi_1 \pi_2 \dots \pi_l = \beta^{-1}(c_1)\beta^{-1}(c_1) \dots \beta^{-1}(c_l)$ *is a concatenation of phase decompositions of different occurrence nets in* CSON. *We also denote, for all* c_j *and* e_j *occurring in the chain* $\xi_{\text{ON}_j^\uparrow}$, $\pi(c_j) = \pi_j$, *and*

$$\mathit{before}(e_j) = \mathit{pre}(\mathit{Max}_{\beta^{-1}(\mathit{Pre}(e_j))}) \times \{e_j\} \cup \{e_j\} \times \mathit{post}(\mathit{Min}_{\beta^{-1}(\mathit{Post}(e_j))})$$

3. *The relation*

$$(\sqsubset \cup \prec \cup \lhd)^* \circ (\prec \cup \lhd) \circ (\prec \cup \sqsubset \cup \lhd)^*$$

over $\mathbf{E} \cup \mathbf{E}^\uparrow$ *is irreflexive, where:*
   - $e \prec f$ *if there is* $c \in \mathbf{C} \cup \mathbf{C}^\uparrow$ *with* $c \in e^\bullet \cap {}^\bullet f$;
   - $e \sqsubset f$ *if there is* $q \in Q \cup Q^\uparrow$ *with* $q \in e^\bullet \cap {}^\bullet f$; *and*
   - $e \lhd f$ *if* $(e, f) \in \bigcup_{e' \in \mathbf{E}^\uparrow} \mathit{before}(e')$.

The initial marking M_0^{BSON} *of* BSON *is the initial marking of the* $\text{CSON}^\uparrow$ *together the initial markings of all the* ON_i*'s such that* $\beta(M_0^{\text{ON}_i}) \cap M_0^{\text{CSON}^\uparrow} \neq \varnothing$.

Definition 5(1) implies that each phase points to exactly one condition of the upper level ON, and Definition 5(2) means that each upper level condition maps to a single phase of a lower level ON. The ordering of the upper level conditions must match that of the phase decompositions of the lower level ON. *before*(e) captures some new causal dependencies between events coming from both levels of BSON. Intuitively, it represents the 'happened before' relationship on the events. In Definition 5(3) it is required that the new dependencies, together with the communication (i.e., $\sqsubset$) and ordinary causal relations (i.e., $\prec$), which are already present in the model are acyclic.

Note that in a BSON, the initial marking of a lower level ON may not belong to the initial marking of the BSON. Such a net may be 'waiting' for the firing of some events in other ONs. For the BSON in Figure 1, $\{c_0\}$ is the initial marking of ON_2 but it is not the initial marking of the BSON. $\{c_0\}$ can be reached only if the 'update' event has happened.

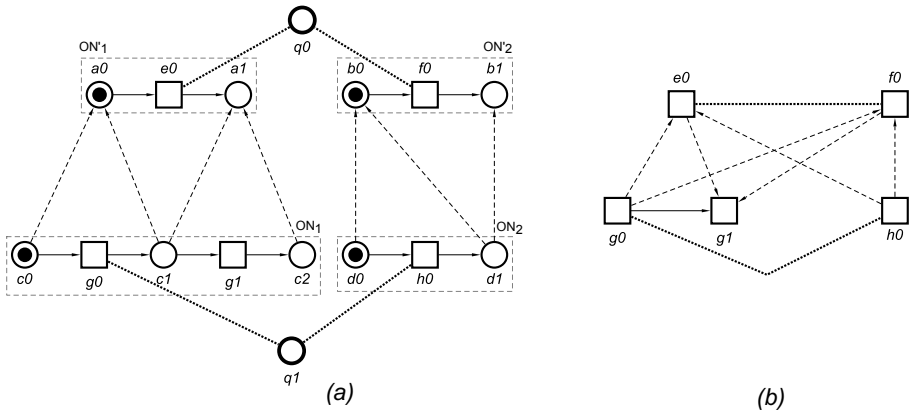

Fig. 5. (a) A BSON with two upper level ONs and two lower level ONs. (b) Three types of relationships over events of (a).

Figure 5(a) shows a BSON example involving synchronous communications in both levels. The lower level CSON consists of two interacting systems, ON_1 and ON_2. An information about their evolution is provided in the upper level by $\text{ON}_1^{\uparrow}$ and $\text{ON}_2^{\uparrow}$ respectively. The initial marking is $M_0^{\text{BSON}} = \{a_0, b_0, c_0, d_0\}$. The related phase decompositions are as follow:

$$\beta^{-1}(C_1) = \beta^{-1}(a_0)\ \beta^{-1}(a_1) = \pi_1\ \pi_2 = \{c_0, c_1\}\ \{c_1, c_2\}$$
$$\beta^{-1}(C_2) = \beta^{-1}(b_0)\ \beta^{-1}(b_1) = \pi_3\ \pi_4 = \{d_0, d_1\}\ \{d_1, d_2\}$$

where C_1 and C_2 are sets of conditions in $\text{ON}_1^{\uparrow}$ and $\text{ON}_2^{\uparrow}$ respectively. One can observe that the succession of the conditions in each upper ON corresponds to a valid phase decomposition in the lower ONs. For the phases $\pi_1\pi_2$ in $\text{ON}_1^{\uparrow}$, we

have $Min_{\pi_1} = \{c_0\}, Max_{\pi_1} = Min_{\pi_2} = \{c_1\}$ and $Max_{\pi_2} = \{c_2\}$. Using the phase information and the relations captured by CSON, we obtain the $before(e)$ relations of two upper level events e_0 and f_0, as follows:

$$\begin{aligned}
before(e_0) &= pre(Max_{\beta^{-1}(Pre(e_0))}) \times \{e_0\} \quad \cup \quad \{e_0\} \times post(Min_{\beta^{-1}(Post(e_0))}) \\
&= (pre(Max_{\beta^{-1}(a_0)}) \cup pre(Max_{\beta^{-1}(b_0)})) \times \{e_0\} \ \cup \\
&\quad \{e_0\} \times (post(Min_{\beta^{-1}(a_1)}) \cup post(Min_{\beta^{-1}(b_1)})) \\
&= pre(Max_{\{c_0,c_1\}}) \cup pre(Max_{\{d_0,d_1\}})) \times \{e_0\} \ \cup \\
&\quad \{e_0\} \times (post(Min_{\{c_1,c_2\}}) \cup post(Min_{\{d_1\}})) \\
&= (pre(c_1) \cup pre(d_1)) \times \{e_0\} \quad \cup \quad \{e_0\} \times (post(c_1) \cup post(d_1)) \\
&= \{g_0, h_0\} \times \{e_0\} \quad \cup \quad \{e_0\} \times \{g_1\} \\
&= \{(g_0, e_0), (h_0, e_0), (e_0, g_1)\} \\
before(f_0) &= \{g_0, h_0\} \times \{f_0\} \quad \cup \quad \{f_0\} \times \{g_1\} \\
&= \{(g_0, f_0), (h_0, f_0), (f_0, g_1)\} \ .
\end{aligned}$$

Thus, we have the following causal relationships (over events) for this BSON:

$$\begin{aligned}
causality &: \prec = \{(g_0, g_1)\} \\
weak\ causality &: \sqsubset = \{(e_0, f_0), (f_0, e_0), (g_0, h_0), (h_0, g_0)\} \\
before &: \lhd = \{(g_0, e_0), (e_0, g_1), (h_0, e_0), (g_0, f_0), (f_0, g_1), (h_0, f_0)\} \ .
\end{aligned}$$

Figure 5(b) illustrates the above relationships diagrammatically. The solid lines represent the causal relations $\prec$; the bold dashed lines indicate the dependencies $\sqsubset$ captured by a/synchronous communications; and the dashed lines represent $\lhd$ relations. Intuitively, the meaning of $\lhd$ is that, for example, g_0 and h_0 must happen before e_0, while g_1 must happen after e_0, since the former two events belong to the pre-phase of e_0 while the latter one belong to the post-phase of e_0. We can observe from the diagram that this BSON satisfies the acyclicity conditions described in Definition 5(3).

Remark 1. A causal cycle in BSON in general involves occurrence nets in both levels. For instance, the model in Figure 6(a) is as in Figure 5(a) except for the synchronous communication in the lower level between g_1 and h_0. Such a model is not a valid BSON structure. The events $\{e_0, g_1, h_0, f_0\}$ form a causal cycle (see the relationships portrayed in Figure 6(b)). It indicates e_0 happens before g_1, and h_0 happens before f_0, but $\{e_0, f_0\}$ and $\{g_1, h_0\}$ must execute simultaneously due to synchronisation. As a result, none of them can be ever be executed. □

Next we define the BSON firing rule which takes care of the marking moving across different phases. Given a marking, there are three requirements to decide whether a step is BSON-enabled: (*i*) it is CSON-enabled (Definition 3); (*ii*) for each upper level event, the maximal conditions in the phase of its input condition is in the current marking; and (*iii*) for each lower level event, its corresponding upper level condition is in the current marking.

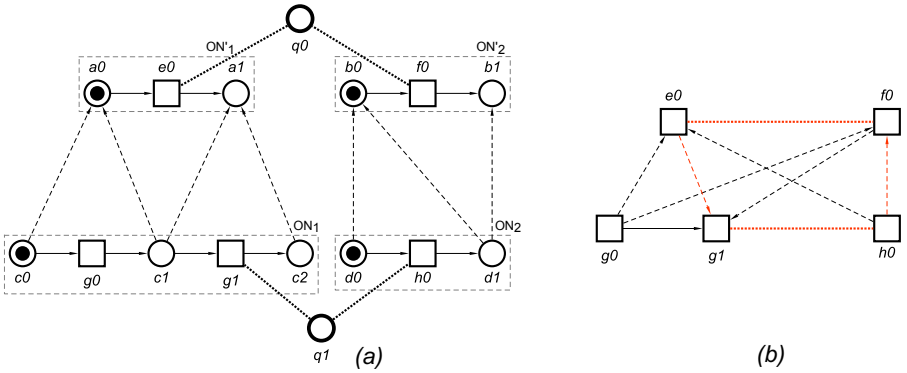

Fig. 6. (a) An invalid BSON which involves a causal cycle (b).

Below we assume that if $e \in E$ is an event in an upper level $\text{ON}^{\uparrow}$ (i.e., $\text{ON}^{\uparrow} = (C, E, F)$), then $\pi({}^{\bullet}e) = \beta^{-1}({}^{\bullet}e \cap C)$ is the phase of the input condition of e, and $\pi(e^{\bullet}) = \beta^{-1}(e^{\bullet} \cap C)$ is the phase of the output condition of e. The markings and steps of BSON are defined similarly to those for CSON.

Definition 6 (BSON firing rule). *Let* BSON *be as in Definition 5.* $M \subseteq \mathbf{C} \cup \mathbf{C}^{\uparrow}$ *be a marking, and* $U \subseteq \mathbf{E} \cup \mathbf{E}^{\uparrow}$ *be a step of* BSON.

1. U *is* BSON-enabled *at* M *if*
   - $({}^{\bullet}U \setminus U^{\bullet}) \subseteq M$, *i.e.,* U *is* CSON-*enabled;*
   - $Max_{\pi({}^{\bullet}e)} \subseteq M$, *for every* $e \in \mathbf{E}^{\uparrow}$., *i.e., ;*
   - $\beta(e') \in M$, *for every* $e' \in \mathbf{C}$.
2. *If* U *is* BSON-*enabled at* M, *then* U *can be* fired *and produce a marking* M' *given by:*

$$M' = (M \setminus ({}^{\bullet}U \cup Max_{\pi({}^{\bullet}U)})) \cup U^{\bullet} \cup Min_{\pi(U^{\bullet})}$$

where $Max_{\pi({}^{\bullet}U)} = \bigcup_{e \in U} Max_{\pi({}^{\bullet}e)}$ *and* $Min_{\pi(U^{\bullet})} = \bigcup_{e \in U} Min_{\pi(e^{\bullet})}$. *This is denoted by* $M[U\rangle M'$.

The definitions of *step sequences* and *reachable markings* of BSON are similar to those for CSONs.

For example, $U_1U_2U_3 = \{g_0, h_0\}\{e_0, f_0\}\{g_1\}$ is a possible step sequence of the BSON in Figure 5: The only step U_1 enabled at the initial marking $M_0 = \{a_0, b_0, c_0, d_0\}$ is $U_1 = \{g_0, h_0\}$ since it is CSON-enabled as well as the corresponding upper level conditions (a_0 and b_0) are marked. The firing of U_1 changes the marking to $\{a_0, b_0, c_1, d_1\}$ which enables the step $U_2 = \{e_0, f_0\}$ (note that the conditions in $Max_{\pi}({}^{\bullet}e_0) = \{c_1\}$ and $Max_{\pi}({}^{\bullet}f_0) = \{d_1\}$ are marked). The firing of U_2 produces $\{a_1, b_1, c_1, d_1\}$ and also enables $U_3 = \{g_1\}$ which produces the final marking $\{a_1, b_1, c_2, d_1\}$.

The following result is a re-statement of Proposition 1. In particular, it explains the consistency between the temporal ordering of events involved in a step sequence and the relations provided by BSON.

Proposition 6. *Given a step sequence of* BSON $\lambda^{\text{BSON}} = U_1 \dots U_n$ *($n \geq 0$), we have that:*

1. *If $i \neq j$ then $U_i \cap U_j = \varnothing$, i.e., no event occurs more than once.*
2. *There is a step sequence involving all the events in* $\mathbf{E}$.
3. *If $i \geq j$ then $(U_i \times U_j) \cap ((\sqsubset \cup \prec \cup \lhd)^* \circ (\prec \cup \lhd) \circ (\prec \cup \sqsubset \cup \lhd)^*) = \varnothing$, i.e., the causal predecessors of an event can never be executed after or together with that event.*

Proof. Similar to that in [7], after suitable adaptation to accommodate refinements introduced in this paper. □

5 Implementation

The visual editing of structured occurrence nets, their simulation, analysis and verification are the functionalities supported by the SONCraft toolkit. The tool is implemented as a Java plug-in within the Workcraft platform, which provides a flexible framework for the development and analysis of interpreted graph models. The platform is built using a plugin-based architecture and supports run-time scripting, which makes it easily extendible to new graph-based formalisms as well as analyses and verification methods. It also provides a GUI environment that facilitates model entry and supports interactive visual simulation, together with convenient "single-click" verification. So far several modules have been implemented and supported by the platform, including structured occurrence nets (SONCraft), Petri nets, and many other Petri net based formalisms, for example, STG [17] and CPOG [10]. A detailed SONCraft and Workcraft description and manuals can be found in [2, 9]. The present version of SONCraft deals with three types of relation that have been defined for SONs, named communication, behavioural and temporal abstractions [3].

This section presents an overview of the major features provided by SONCraft. We also describe algorithms implemented in the analysis tools.

5.1 Visualisation

The graphical interface of SONCraft is depicted in Figure 7. The *Main menu* provides the functions to manage, edit and analyse models. For example, the *Tools* menu provides a set of user-friendly analysis tools for model checking; and there is a vector graphics export function in *File* menu (all the SON models shown

[3] Temporal abstraction (TSON) is used to define atomic actions in a system, i.e., actions that appear to be instantaneous to their environment. A detailed description of TSON can be found in [7].

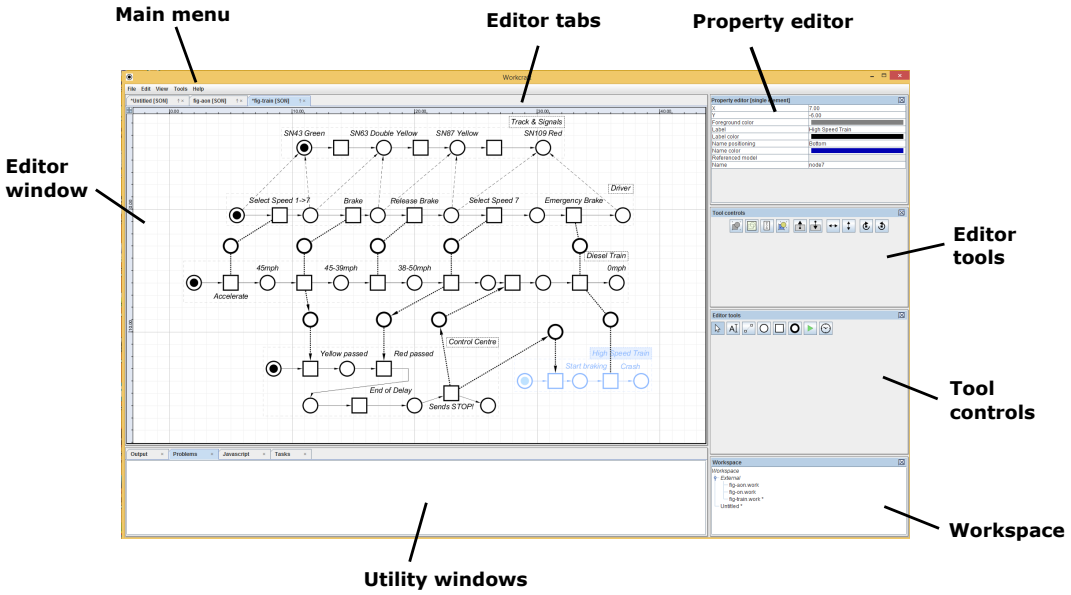

Fig. 7. SONCraft interface

in this paper were imported directly from SONCraft with minimal modification). The *Editor tabs* line shows the names of all of the opened models and allows the user to choose which one is to be displayed in the *Editor window*, which is the place for the user to create, edit and simulate a SON model.

SONCraft defines a series of graphical nodes and connection types displayed in the *Editor tools*, allowing user to create and edit SON-based models. The *Property editor* panel at the top-right hand side is used to support various visual node editing operations, e.g., to change the label, color or position of a condition. The *Tool controls* panel provides access to the extended functionality of a selected tool. For example, when the connection tool is activated, the user is able to switch between causal and behavioural connections in order to construct different types of SON abstractions. The *Workspace window* lists opened or imported work files. One can also operate on a work file (delete, save, etc). The *Utility window* is used for showing additional information concerning the progress of currently executed tasks, verification results, and information about any errors that may have occurred during execution.

5.2 Structural property checking

SONCraft provides the user with a set of structural verification algorithms that can be used to validate the model. It is important to verify the correctness of

Algorithm 1 (BSON cycle detection)

Inputs:
BSON — behavioural structured occurrence net
Output:
Result — causal cycles

```
convert BSON to G = (V, E).
for all e ◁ f do
    add (e, f) to E
Result = tarjan(G)          # compute SCCs of G
filter(Result)

function filter(Result)
for all SCC ∈ Result do
    if SCC.size == 1 then
        remove SCC from Result
    else if !SCC.contain(≺ ∪◁) then
        remove SSC from Result
```

structure before further analysis, otherwise the results are likely to be incorrect. The verification criteria follow from the formal definitions and properties introduced in this paper and [7].

The *Relation property checker* deals with the correctness of basic relationships and structure in a SON model. The algorithms it uses include, for example, conflict-freeness checking, phase decomposition checking, and component ONs disjointness checking. The *Acyclic property checker* focuses on the acyclicity condition of SONs. The verification of such a property comes down in practice to searching strongly connected components (SCC) in a SON model. The checker applies Tarjan's algorithm [16] to compute maximal SCCs, and then uses a filter to obtain the desired results. As an example, Algorithm 1 carries out acyclicity checking for BSONs. The algorithm first converts a BSON to a graph $G = (V, E)$, where V is the set of nodes including all conditions, events and channel places of the BSON, and the set of E is the arcs representing all causal relationships and weak causal relationships. The algorithm then computes *before*(e) for every upper-level event as additional relations for the input graph. The *filter*() function at the end of the algorithm aims to remove all the cycles which only involve weak causality, i.e., sync-cycles.

5.3 SON simulator

SONCraft offers a built-in simulator for ONs, CSONs, and BSONs. The underlying semantics of SON-based simulation follows the firing rules presented above. The simulation function in SONCraft can be activated by clicking on the simulation button in the editor tools panel. The initial marking will be automatically set, i.e., all the input conditions of all the ONs will be filled with black tokens (except for those ONs that are 'waiting' for an event in another ON) indicating the

start points of the system. Moreover, all enabled events will be highlighted. The simulation can then be conducted either manually or automatically, by firing a succession of enabled events, causing tokens to move, event highlighting to be updated, and the simulation record augmented.

Algorithm 2 (Computing CSON-enabled step)

Inputs:
CSON — communication structured occurrence net
M — current marking
Output:
U — a step CSON-enabled at M

```
1: U = ∅
2: Del = ∅                          # deleted events
3: for all e ∈ E do
4:    if •e ⊆ M then                # e is ON-enabled
5:       add e to U
6: for all e ∈ U do
7:    if e ∉ Del then
8:       min = minParallel(e)
9:       for all f ∈ min do
10:         if f ∉ U then           # minimal parallel firings of e is not ON-enabled
11:            add all events in min to Del
12:            break
13: U = U \ Del

14: function minParallel(input: e)
15: Result = ∅
16: mark e visited
17: add e to Result
18: for all g such that g ⊏ e do
19:    if g is unvisited and q ∉ M, where q ∈ g• ∩ •e then
20:       add all events in minParallel(g) to Result
21: return Result
```

The procedure for computing (minimal) CSON-enabled steps is given in Algorithm 2. The idea is to first compute a step U including all the ON-enabled events of CSON in order to narrow down the size of the search, and then remove the events which do not meet CSON-enabled requirement from U.

Unlike the execution of a standard occurrence net, where a step sequence can be composed out of a sequence of single events firings, in CSON there may exist *minimal parallel firings* for an event, where one enabled event implies all events in the minimal parallel firings are enabled as well. Both $\{f_1, e_1\}$ in Figure 3 and $\{e_0, f_0, g_0\}$ in Figure 4 are such steps because of their synchronised behaviour. Note that such minimal parallel firing can involve either synchronous (see Proposition 3) or asynchronous communications (see Proposition 4). There-

fore, in the algorithm it is not possible to only consider the enabling for a single event. Instead, all its minimal parallel firings are considered in the computation.

The pseudocode for computing minimal parallel firings of a given event is presented in function *minParallel*. The function uses a working list *Result*, initialized to the given event. Then it recursively visits the weak causal predecessors of the node in the list. The predecessor can be added to the working list if it is unvisited and the channel place between the two events is unmarked.

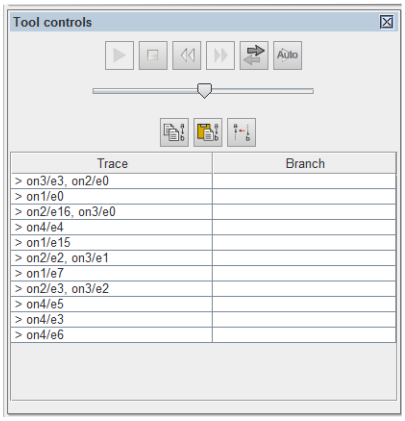

Fig. 8. Simulation control panel

The simulation tool control panel provides access to several additional simulation functions, most of which relate to the simulation traces which are recorded during the simulation (see Figure 8). For example, the *Playback* button is used to automatically playback an existing trace, at a selectable speed; the *Reverse/Forward simulation* buttons are used to change the simulation directions; and the the *Automatic simulator* control causes simulation to occur, using maximum parallelism through to the end.

Error tracing : The SON simulator provides a failure analysis function called error tracing. When the failure analysis function is on, each event has an associated *fault bit* '1' or a '0'. This bit can be used to indicate whether one wishes to regard the event as a faulty one, with '1' indicating a simulated fault. An error count is also shown below each condition, and is set initially to '0'. This count cannot be changed manually by the user. Rather it is automatically calculated during simulation to indicate for each condition the number of faults that have been passed on the forward route to the condition.

5.4 Reachability checking

Once a SON model is complete and its structure is valid, the user can perform model checking. SONCraft provides a reachability checker for verifying reachability. Such analysis establishes whether a given marking, i.e., a set of conditions and/or channel places, can be reached from the initial marking. SONs are acyclic (without causal cycles), conflict-free (no alternative behaviour is allowed) and 1-safe (a condition/channel place can contain at most one token). It has been proved that the reachability problem in this subclass of Petri nets turns out to be linear [4].

Given a set of required conditions and channel places, the SON reachability algorithm proceeds as follows (see Algorithm 3 for details):

1. compute all the causal predecessors of required nodes (e.g., the relations presented in Figure 5(b));
2. check that none of the required nodes are consumed by (the input of) their causal predecessors;
3. check that none of the corresponding upper level conditions (w.r.t β) of the required node are consumed by their causal predecessors.

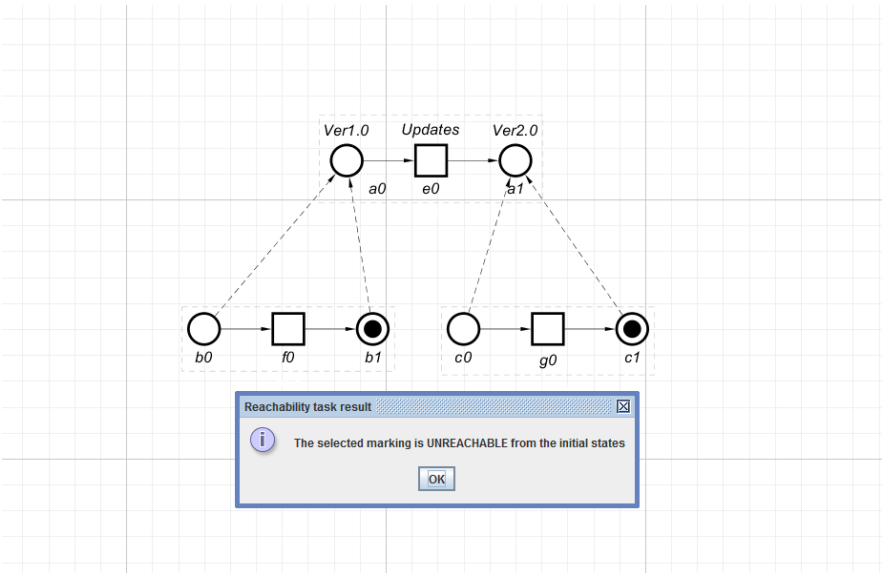

Fig. 9. Reachability task result for a SON model with two marked conditions.

The computation of causal predecessors in step 1 takes into account all three types of causal relations in ONs, CSONs and BSONs. The procedure *Predecessors*

Algorithm 3 (Reachability checking)

Inputs:
SON — Structured occurrence nets
M — Marking of SON

Output:
Whether M is reachable from the initial marking

```
1: Pred = ∅                    # all predecessors of M
2: Cons = ∅                    # input conditions and channel places of events in Pred
3: for all c ∈ M do
4:    Predecessors(c)
5: for all n ∈ Pred do
6:    if n is an event then
7:       add all nodes in •n to Cons
8: for all c ∈ M do
9:    if c ∈ Cons ∨ Cons contains all β(c) then
10:      return FALSE
11: return TRUE

12: procedure Predecessors (input: c)
13: mark c visited
14: add c to Pred
15: for all c' ∈ CausalPreset(c) do
16:    if c' is unvisited then
17:       Predecessors(c')

18: function CausalPreset(input: c)
19: Preset = ∅
20: for all node c' such that (c', c) ∈ F ∨ (c', c) ∈ W do
21:    add c' to Preset
22: for (e, f) ∈ ◁ do
23:    if f == c then
24:       add e to Preset
25: return Preset
```

is recursively called for exploring causally related nodes of M in the backwards direction. If a node is visited twice or a condition is initial state, then the procedure reaches the stop condition. Set *Pred* is used to store all causal predecessors during the exploration. Step 2 concerns the basic reachability criterion in SONs. That is, M is unreachable if there exist two nodes in M such that one causally precedes the other. Step 3 addresses the consistency between the marking in different levels, i.e., M is unreachable if there is an upper level condition c in M and a lower level condition c' in M such that c causally precedes $\beta(c')$.

Figure 9 shows a simple reachability task result reported in SONCraft. The causal predecessors of the marking $\{b_1, c_1\}$ are $\{f_0\}$ and $\{e_0, f_0, g_0\}$ respectively. This marking is unreachable from the initial marking (as shown in the dialog). This is because the upper level condition of b_1 (viz. a_0) is consumed by one of the

causal predecessors e_0. Intuitively, the unreachability follows since $\{a_0, b_1\}$ will change to $\{a_1, c_0\}$ after firing e_0. In the case of the verified marking is reachable, then a request can be made for the trace leading to the marking to be passed to the simulation tool for playback or further analysis.

5.5 Tool Architecture

SONCraft is written in JAVA making it accessible on all platforms for which there exists a JVM. The architecture depicted in Figure 10 shows a detailed view of the integration between the Workcraft framework and SONCraft.

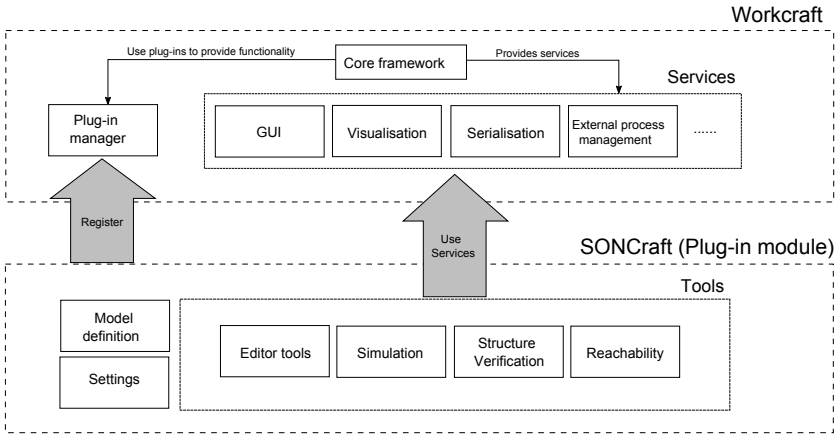

Fig. 10. Tool architecture.

Workcraft architecture The Workcraft framework consists of the following three parts:

The *Core framework* is in charge of the initialisation of Workcraft, managing plug-ins and provision of common services to the plug-ins. When the program starts up, services such as the configuration manager and the framework GUI are initialised. This is followed by the initialising of the plug-in manager, which provides the facility for loading all existing plug-ins. On shut-down, Workcraft saves the configuration of the framework; it restores it on the next start-up.

The *Plug-in manager* is responsible for scanning and loading all plug-in modules which have been registered to the manager. A plug-in module is a related collection of plug-ins that together implement a specific functionality, for instance the SONs modeule. For each plug-in module, the manager also maintains a list of its internal facilities. During initialisation the plug-in manager uses the

list to load the contents of plug-ins instead of scanning the plug-ins directory every time.

The *Services* are fully managed by Workcraft and accessible to the plug-ins. The GUI service provides the facilities for creating editor, tool and information windows. A number of advanced GUI capabilities, such as the multiple document interface and full-screen mode, are also supported. The Visualisation service facilities provide editing functions for the node types defined by any model, for instance, drawing, transformation and auxiliary editing operations. The Task management service is responsible for executing all external process tasks – it maintains the list of all running tasks and uses a separate thread for parallel execution.

SONCraft integration SONCraft is deployed in the Workcraft framework as an individual plug-in module. There are three main components inside the module:

The *Model definition* component describes the basic features of a SON model. The component is divided into mathematical and visual levels in order to avoid mixing unrelated responsibilities. The mathematical model describes all the semantics concerning model integrity — it keeps information such as connection types, node names etc. The visual model is a manageable interface between user and both the mathematical and the visual models. The visual model defines how to draw/present SON models as well as maintaining visual information, such as colour, position, label, etc.

The *Settings* component records default properties of a SON model and stores them in a configuration XML file. The Workcraft start-up process loads the stored settings and allows other components to read their configuration variables.

The *Tools* component manages all the external and built-in tools in SONCraft. The implementation of each component tool uses the services provided by the Workcraft framework. For example, the editor tools and simulation facilities rely on the GUI and visualisation services for node placement, trace table creation, etc. Structural verification and reachability checking invoke external process management for monitoring and managing the tasks.

5.6 Installation

The latest version of SONCraft is available from [1]. It is necessary to have a compatible Java Runtime Environment (JRE) version 7 or higher in order to run SONCraft [4]. There is no automatic installer for SONCraft; to install it, the files from the link archive need to be extracted manually. A comprehensive user manual can be found in [9].

[4] JRE download: http://www.oracle.com/technetwork/java/javase/downloads/

6 Conclusions

In this paper, we have discussed structured occurrence nets. The execution semantics for each variant of SONs has been defined. For CSONs, we introduced and investigated the notion of a channel place which is a flexible way to represent asynchronous and synchronous communication. In particular, synchronous communication does not only exist between two events, but may also involve multiple events in different occurrence nets. This led to the notion of a sync-cycle. For BSONs, a refinement of the relation $before(e)$ has been proposed which captures causality between upper and lower level CSONs using causal dependencies between events. In the original definition, such a dependency is captured by conditions which may sometimes produce undesirable effects.

Section 5 introduced the SONCraft tool-kit for construction, simulation and verification of SONs. SONCraft provides a user-friendly graphical interface enabling the user to construct models easily and quickly. The tool offers a powerful simulator and a set of analytical tools. A detailed description of how to use, together with the downloading and installation instructions, can be found in the user manual [9].

An interesting and practically important extension of SONs would be a support for alternative behaviours. Such an extension would make it possible to model and analyse more complex evolving systems, e.g., complex (cyber) crimes or an major accident are both likely to result in a mass of contradictory or uncertain evidence. There has already been some investigations concerning the enhancement of SON for such situations, in fact so as to portray multiple alternative behaviours. [15] discusses the basic idea and outlines a formalisation of communication alternative SON. [6] introduces a system-level counterpart of CSONs built out of the Place Transition nets. [8] addresses the concept of CSON's high level net unfolding which can be regarded as an underlying model of CSONs with alternative.

Finally, work has also started on adding time and probability information to SONs, and on extending the current implementation of the tool-kit accordingly.

References

1. Soncraft homepage, `https://soncraft.codeplex.com/`
2. Workcraft homepage, `http://workcraft.org`
3. Best, E., Devillers, R.: Sequential and concurrent behaviour in petri net theory. Theoretical Computer Science 55(1), 87–136 (1987)
4. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. In: Shyamasundar, R.K. (ed.) FSTTCS. Lecture Notes in Computer Science, vol. 761, pp. 326–337. Springer (1993)
5. Janicki, R., Koutny, M.: Invariants and paradigms of concurrency theory. Future Gener. Comput. Syst. 8(4), 423–435 (Sep 1992)
6. Kleijn, J., Koutny, M.: Causality in structured occurrence nets. In: Dependable and Historic Computing. vol. 6875, pp. 283–297. Springer Berlin Heidelberg (2011)

7. Koutny, M., Randell, B.: Structured occurrence nets: A formalism for aiding system failure prevention and analysis techniques. Fundamenta Informaticae 97(1), 41–91 (Jan 2009)
8. Li, B., Koutny, M.: Unfolding cspt-nets. In: PNSE @ Petri Nets 2015. pp. 207–226 (2015)
9. Li, B., Randell, B.: Soncraft user manual. Tech. Rep. CS-TR-1448, School of Computing Science, Newcastle University (Feb 2015)
10. Mokhov, A., Yakovlev, A.: Conditional partial order graphs: Model, synthesis, and application. Computers, IEEE Transactions on 59(11), 1480–1493 (Nov 2010)
11. Poliakov, I.: Interpreted graph models. Ph.D. thesis, School of Electrical, Electronic and Computer Engineering, Newcastle University (2011)
12. Poliakov, I., Khomenko, V., Yakovlev, A.: Workcraft–a framework for interpreted graph models. In: Applications and Theory of Petri Nets, pp. 333–342. Springer Berlin Heidelberg (Jun 2009)
13. Randell, B.: Occurrence nets then and now: the path to structured occurrence nets. In: Applications and Theory of Petri Nets. pp. 1–16. Springer Berlin Heidelberg (Jun 2011)
14. Randell, B., Koutny, M.: Failure: their definition, modelling and analysis. In: Theoretical Aspects of Computing–ICTAC 2007. pp. 260–274. Springer (Sep 2007)
15. Randell, B., Koutny, M.: Structured occurrence nets: Incomplete, contradictory and uncertain failure evidence. Tech. Rep. CS-TR-1170, School of Computing Science, Newcastle University (Sep 2009)
16. Tarjan, R.: Depth first search and linear graph algorithms. SIAM Journal on Computing (1972)
17. Yakovlev, A., Lavagno, L., Sangiovanni-Vincentelli, A.: A unified signal transition graph model for asynchronous control circuit synthesis. In: Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design. pp. 104–111. ICCAD '92, IEEE Computer Society Press, Los Alamitos, CA, USA (1992)